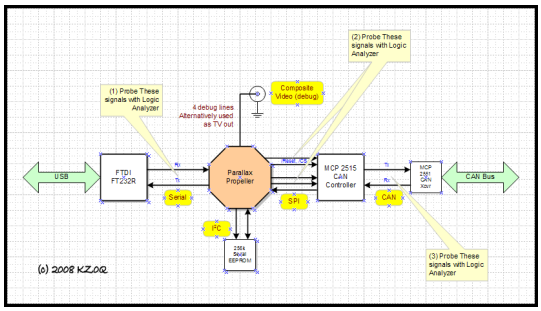
turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

MONDAY, JANUARY 7, 2008, 22:43 MST

AND SO IT STARTS...



Ok, I've a lot to do to enable this device. It's basically working but it lacks performance in many areas. I'll eventually describe each area and see if we can address them one by one. Hey, if you have constructive ideas, "I'm all ears". ;-)

I hope you enjoy this trek...

Oh, yes... what is propCAN? The diagram (click to enlarge) shows the major blocks of this device, the protocols being used and which signals are probed by the logic analyzer so we can verify the software operation.

For device pictures (my hand soldering of surface mount parts ;-), board layout, logic analyzer probe attachment, and further device description see: http://propcan.moraco.us/

Next up... let's talk general design goals followed by software organization (with links and pics, of course)

#POR - Plan of Record

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

TUESDAY, JANUARY 8, 2008, 18:18 MST

LET'S TALK PERFORMANCE EXPECTATION

When I started this project expecting to use the Propeller chip and the FTDI chip I had to "run some numbers" to see if this could be done.

One aspect of using the Propeller chip is that there is a lot more software simulation of communication than with other microcontrollers. In the Propeller there is no USART, no SPI or I2C hardware. We have to do all of this in software. There are rudimentary proof-of-concept objects available for Spin (the propeller programming language) but they do not get near the speeds (in bytes per second throughput) that we need. How do I know what I need? Let's look at the CAN bus traffic to determine this.

This USB to CAN controller is for general CAN use but is being created so we can control or interact with a "constellation" of "Widgets" on the CAN Bus. See http://can-do.moraco.info/ for a description of the "Widget" including full user manual and Programmer's Guide. This Widget is designed to be the point of integration for each payload to the satellite's house-keeping computer. This "constellation" of "Widgets" is really the collection of payloads in the satellite all communicating with the house-keeping computer over the CAN Bus.

The protocol for communication with these Widgets is a small subset of the possible range of CAN messages. The actual messages used are called out in full detail in the appendix of the User's Guide at the aforementioned web-site.

In order to "run the numbers" I had to select the most useful way to view the traffic. That is CAN messages consist of a header and an optional payload followed by a few more fields (checksum, ack, etc.). The data portion of the message contains "Stuff bits" which are bits injected by the transmitter into the message so that there is never more than 5 consecutive ones or zeros. This facilitates clock recovery from the serial data stream. You can read in more detail in the CAN spec. but the important part for us is that there is a minimum bit length and a maximum for each payload size (0-8 bytes). For performance measurement then I chose to use the minimum (which we mostly can't attain) so that we'd get best saturation of the CAN Bus.

We have 0-byte, 2-byte and 8-byte payloads in the AMSAT protocol so my numbers yield the following as goals:

AMSAT CAN Bus rate: 800k bps (800,000 Hz)

Minimum bits lengths for CAN Messages:

- 0-byte payload: 47 bits for a max rate of 17,021 messages/sec
- 2-byte payload: 63 bits for a max rate of 12,698 messages/sec
- 8-byte payload: 111 bits for a max rate of 7,207 messages/sec

Oddly enough these numbers inform our serial (propeller to FTDI USB chip) data-rates by the length of the 8-byte payload packets and inform our SPI (MCP2515 to propeller) data-rates by the max rate of zero-byte payload message arrivals from the CAN Bus. These numbers are then:

- Max Serial Tx rate: 1.51 Mbits/sec (10bit characters at 22 chars/message)
- Max SPI rate: 1.8 Mbits/sec (17,021 messages per second at 14 bytes per message) [actually this might be slightly higher based on SPI transactions used]

So, there we are. We have pretty high maximum data rates for both Serial Tx and SPI.

Now, let's move on to discussing the first firmware functional organization.

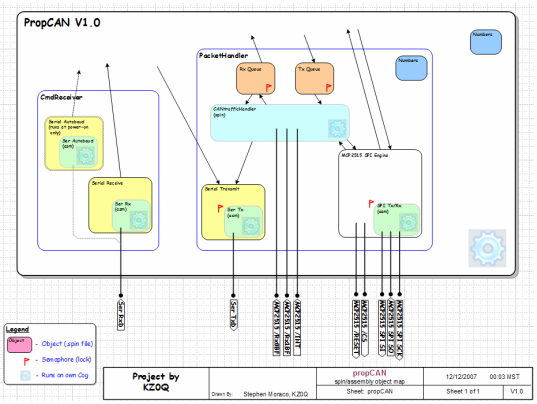
#Performance

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

TUESDAY, JANUARY 8, 2008, 22:19 MST

WHICH COG IS DOING WHAT?



The propeller is "fun" in that we have eight identical CPU's on the one chip. So how should the work be divided amongst them? That's what this post is about.

Each CPU is called a Cog. As I'm starting this effort I'm attempting to smartly allocate just enough Cogs to create a reasonable separation of function while leaving enough Cogs free to activate monitoring/debug functions in the unused Cogs.

The picture on the right (click for full-size view) shows my current draft functional breakdown and marks Cog assignments with the Blue Gear icon (each blue gear on the diagram means a different Cog is assigned to accomplish that function.) Major I/O pins are shown routed to these Cogs as well to show that each Cog interacts with a different part of the external hardware.

You can also see that I use queueing (Tx and Rx Q's) to stage traffic for handling by the various Cogs.

Finally, please note that I reuse one Cog by first loading the auto-baud detection code into the Cog and then when we know the Serial baud-rate set by the driver communicating with the propCAN device then the Cog is stopped and the serial receive code (or another task) is loaded into the same Cog since the auto-baud code is no longer needed.

So, here you have the initial proposed software organization. This mechanism is basically working today. However, Now we've got some issues to which we need to attend.

In the next post I'll show measurement of the current through-put as seen from a logic analyzer which is watching serial Tx/Rx, SPI bus and CAN Tx/Rx and some extra pins used for debug output.

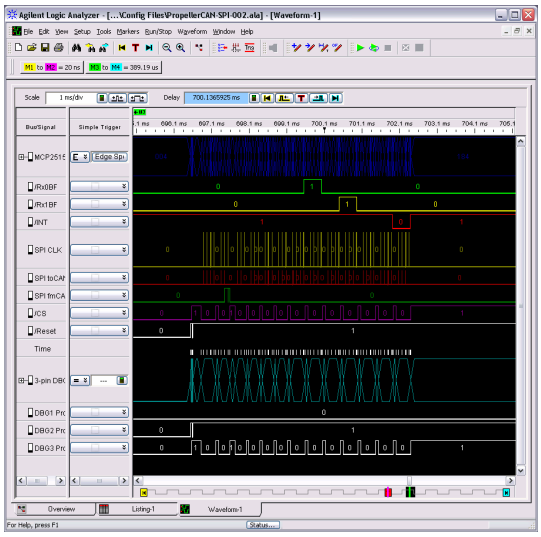
#Software Organization

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

WEDNESDAY, JANUARY 9, 2008, 22:14 MST

TURNING ON THE SPI ENGINE COMMUNICATION WITH THE CAN CONTROLLER



With the firmware organization figured out we start to cobble together the first pieces of code which others have developed for us to test and learn from. In this case I'm starting with the SPI Engine 1.0. If you remember from the first post where I show which lines are probed you know that I've probed the SPI bus and a few extra lines to/from the CAN Controller. The logic analyzer capture (double click to enlarge) shows many bytes being sent to the MCP 2515 which include commands for toggling the /Rx0Bf, Rx1Bf, and /Int lines so we can see that the lines assert when the MCP 2515 wants to signal the Propeller that it needs servicing.

From top down the diagram shows the following signals:

- /Rx0BF
- /Rx1BF
- /INT
- SPI CLK
- SPI data to CAN
- SPI data from CAN
- /CS
- /Reset
- (the remainder can be ignored.)

A quick key to reading this is when the violet line is low (/CS asserted) we are commanding or reading data from the MCP 2515. The dark-red line is data from the propeller to the CAN device, while the dark-green line is data coming from the CAN device. At the top in bright red/yellow/green are the handshake lines coming from the CAN controller which when controlled by the MCP 2515 means something specific. In this test however I've set them up as general purpose I/O so I can toggle them and prove them to be working.

I'm seeing expected wiggles on all the right lines so for now (many software iterations to get to this point) we are having a great day!

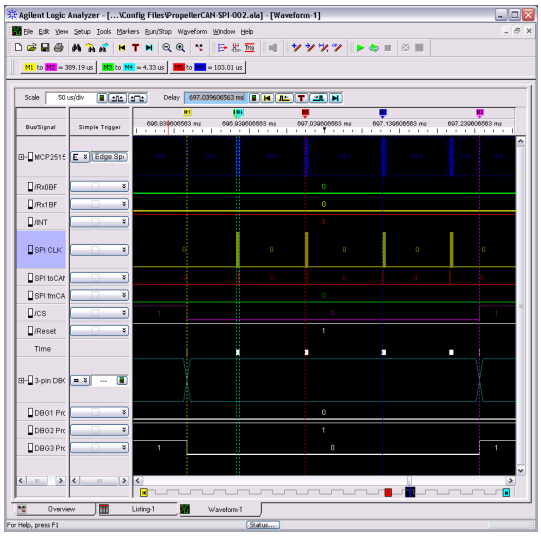
#First Wiggles, Logic Analyzer

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

WEDNESDAY, JANUARY 9, 2008, 22:43 MST

HMMM, NOW THAT IT IS WORKING WHAT DO I REALLY HAVE?



Ok, now we have the SPI engine running. Now let's measure the performance to see how close it is to what we need.

In this view of the traffic I've zoomed in to one SPI transaction. That is, we assert /CS, send one or more commands, maybe read data and then we de-assert /CS.

Referring to our violet /CS line you see that I've zoomed in so that /CS asserted is about the whole width of the waveform window. You'll also see that I've added a few markers (colored vertical dashed lines) denoting signal rising/falling edges of interest. At the top of the waveform you see three measurements in micro-seconds. Let me discuss each of these and what these times now tell us.

Let's work our way in from out-side in. If you look at the /CS line you'll see that I've placed a marker (M1, yellow) at the falling edge (signal asserted) and (M2, violet) at the rising edge. The first measurement at the top shows that M1 to M2 = 389.19 uS which shows, fairly precisely how long /CS is asserted.

Now this is a simple transaction. I'm asserting /CS, writing 4 bytes to the MCP 2515 and then de-asserting /CS.

The starting SPI engine code is built to do one byte at a time handing each byte from one Cog to the assembly language back-end Cog. To measure how long it takes to write a single byte, let it complete and then write another I've placed another two markers: (M5 - Red, and M6 - Blue). The measurement at the top in this case shows M5 to M6 = 103.01 uS. So now we know that handing the byte to the assembly engine waiting for it to be sent and then waiting for the next that it takes ~100 uSec per byte.

Now, I wanted to ask one more question of this waveform. How long does it take to send the one byte once the Assembly back-end Cog starts to send it? In this case I've added two more markers (M3 - green, and M4 cyan). The measurement at the top for these shows M3 to M4 = 4.33 uS.

So, in this case I'm seeing a bit-rate (within byte) of 1/4.33uS or ~225 kbits / sec.

However, since our byte-rate is much slower this degrades to 8-bits / 103 uS or 1/(103/8) = 75.8 kbits / sec.

There are a couple of issues we'll have to address now that we've looked at this performance:

1. Handing one byte at a time between Cogs will not allow us to meet our desired performance given what we're seeing.
2. Looking at the assembly code we can dramatically increase it's bit time performance if we move to a specialized driver (not the general demonstration driver we are starting with.)
   The assembly is built to handle many variations of SPI, once we target to one device we can remove this all purpose code in favor of only enough code to do exactly what we need.
3. A first attempt can be made to move the boundary to a single MCP2515 command but we will likely need to go all the way to handling full transactions in the assembly code not just single commands, we'll see.

NOTE: as a result of this study I came up with and posted the MCP 2515 SPI Engine specialized object: http://obex.parallax.com/objects/227/ which does indeed show that I'll have to go to full transactions in the back-end Cog but that's material for another upcoming post...

In the next post I'll fall back to the initial device turn-on and then move on to aspects of serial Tx and Rx.

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

WEDNESDAY, JANUARY 9, 2008, 23:58 MST

REMEMBERING THE FIRST DAY...

Earlier I stated that working with the propeller was "fun". I think back to first power-on of this device (late Nov 2007, Rev A1 board). The night before, I had completed the soldering of all the parts, studied the board for bad/missed solder joints... and then I cleaned and dried the board.

After letting it sit over-night I finished work, came home and grabbed the new board and cabled it up (plugged it into USB on my Dell XPS Gen4.) I keep the audio on when I'm working with new USB devices so I heard the now overly-familiar "bing" when the new device was added and I saw that the FTDI chip was recognized. Watching my new device I happily see the USB Tx/Rx led's flashing. All of a sudden I stop. "What code can I run to exercise something?"

After a moment of thought, I remembered the simple code for LED flashing found in the Propeller Manual so I reached for the book and started thumbing through it. After a little bit of hurried keying and looking up to which ports I attached the error and warning LEDs I then had some code which should work.

About this time my son walks in about to ask me something (its always something technical but some topic which I never seem to able to predict. ;-) I hold up my hand in a "please-wait" plea and I tell the Propeller tool to download the code to RAM and run it.

Both of us were amazed when it did and then it really did. It found the device and downloaded and verified the code and then it really did toggle the LED! Now that's fun! Having the board turn on so quickly!

In the midst of this "rush" of success I then pressed ^F11 to download to EEPROM. Again, it worked! I gotta say, for my own cobbled together schematic, doing board layout after carefully choosing parts, order the parts and boards, cutting out one of the boards from the rest in the panel and then soldering all the parts and then to see it "just work" I was stunned. This all happened so fast that I wasn't prepared to do anything next. I turned around to my Son we both reveled in how fast this came up and then I talked to him about subject he needed to address when he came into my work-room in the first place.

So, yes this propeller is fun! Many aspects of it are new (e.g., architecture, tool-set, microcode-like assembly language) and it's easy to interface devices to it. What a great "day two" with the Rev A.1 board! Thanks Parallax, this is great fun!

#Day Two, Fun, Initial Turn-on

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

THURSDAY, JANUARY 10, 2008, 00:28 MST

TURNING ON SERIAL -OR- WHO CHANGED THE BAUD-RATE?

PropCAN is a self-contained little box (roughly 2.5" by 1.5" by 3/4") with a DE9P connector at one end for CAN and a mini-USB connector at the other. Oh, and some LEDs show at various places in this little box, too.

Why am I mentioning this? Well it has to just work when it is plugged in to USB for the first time. And I have found an issue which I didn't expect. (Ok maybe I should have or not, I'm not really sure.)

In my first tests of serial I setup the separate Tx and Rx drivers and setup the baud rate to a fixed default (kind-of mid-range). I then set my terminal program to that baud rate and after a few false starts and measuring of baud seen by my analyzer (remember I'm probing the serial lines, too.) I started working and all was well and I made good progress on implementing my command handler (the ASCII command set is presented in the draft manual found at the propCAN web-site).

Then on some sort of whim I changed the baud rate at my terminal program and... wait... hmm... nothing serial is working now! Huh?

I change it back it works. I move to the new rate, it doesn't. What is going on? It always seems to take me a few moments to get over the initial shock of these events. Finally I remember that I am already setup to measure the serial data to see if I'm even still sending data to the device.

So I take a trace and set my markers and then I calculate the baud-rate I'm now seeing -and- I have to pause and recheck. Sure enough, the FTDI chip is actually being affected by the baud rate selection the software is configuring. When I change the baud rate in HyperTerm it changed the back-end baud-rate of the serial Tx/Rx lines coming out of the FTDI chip and going to the Propeller. I totally was not expecting this behavior. Usually, in my experience, once a USB device is involved, the baud rates become meaningless. In this case I now have a device which can be any of the FTDI supported baud-rates when the propeller powers on. Hmmm...

Now you know the origin of the Auto-baud spin object in my software organization chart (earlier post). The manual simply now says that when first connecting propCAN one must first send a couple of carriage-returns before any other traffic so that propCAN can measure the currently configured baud-rate and set itself up appropriately.

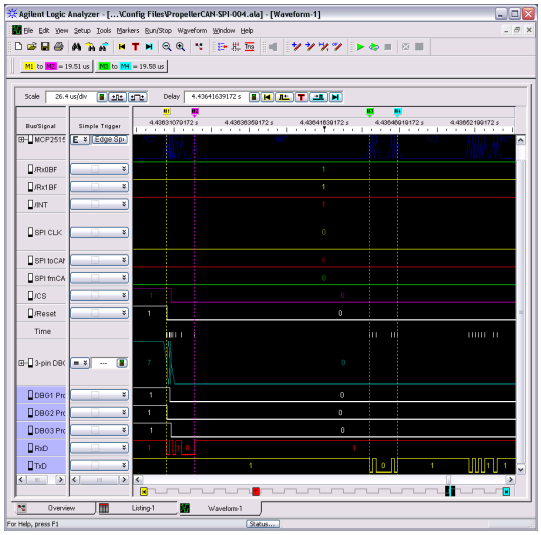
Next post? Let's go over the Auto-baud turn-on (with analyzer trace) and then we'll get back to the New MCP 2515 SPI object turn-on.

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

THURSDAY, JANUARY 10, 2008, 22:54 MST

PROVING WORKING AUTO-BAUD (SERIAL RATE DETECTION)



In the prior post, we learned why we needed automatic baud-rate detection (AutoBaud). In this post, we show the measurement of the new working AutoBaud spin object and we discuss how it works.

The captured analyzer trace (click to enlarge) shows one character arriving via the serial Rx line from the FT232RL and then two characters being transmitted (after a short delay) in response over the serial Tx line. If all is working, we should see that the bit widths are roughly the same.

When the propCAN first starts it loads the AutoBaud Cog and waits for it to make a measurement and return the results. Then with the measured baud rate in hand the AutoBaud Cog is stopped and the serial Rx and serial Tx Cogs (1 each) are started at the measured baud rate.

You may notice (if you have had to spend any time looking at serial data) that the character being received (red line in waveform) is a carriage-return (<CR>, hex $0D, binary %00001101).

Serial characters are sent least-significant bits first with a preceding start-bit and followed by a trailing stop bit. (In this case we are not using parity and one stop-bit, not two.) Therefore, we should see (in waveform order) %0101100001; our ten bits. Also, since the receive (Rx) line is idle at "1" (or high), the trailing one is represented by the final rising-edge and a single bit time thereafter.

This $0d pattern is an "excellent" pattern to use for a number of reasons the first of which is it is easy to remember. From a bit-width discovery perspective, in this one character we've got a start bit, followed by two opposite-polarity single bit times followed by a double-bit time and finally ending with a quad-bit time. Gosh how accurate do we want to get? ;-)

In my case the initial AutoBaud object just measures the width of the least-significant bit, the $01 bit, of the $0d character. Then it looks up the measured value in a table of values representing the extended set of rates that the FT232R will do and picks the closest one. The table is preloaded with calculated values based on the clock frequency at which the Cog is running. I've offset the values by some small percentage in order to increase the capture-range of the measurement. I used a spread sheet to double-

check that my offset-ranges did not create any overlap (at the highest bit rates, smallest values in the table) so that I don't accidentally pick a rate just above or below the desired.

Let's look back at the diagram. M1-yellow and M2-violet show the width of 9 of the ten bits (remembering that the trailing rising-edge is just the start of the stop-bit). We show these nine-bits to be 19.151 uSec wide or 2.166 uSec / bit. This is 461,301 bits/sec. which we know to be the 460,800 bps setting of HyperTerminal.

Moving to the Tx side now we see that M3-green, and M4-cyan mark the first 9-bits of the transmitted response. This is measured at 19.58 uSec or 2.175 uSec / bit. This is 459,652 bits/sec. which, again associates to our 460,800 so we see that our AutoBaud routine has in-fact chosen the correct baud-rate.

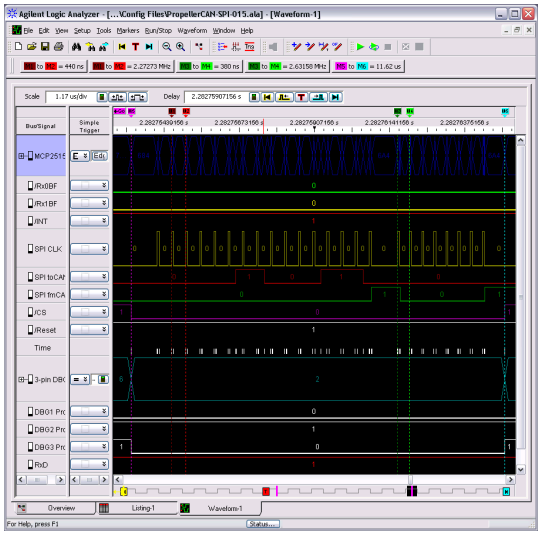
The command language for the propCAN says that when a carriage-return <CR>, $0D, is sent it simply responds with one. This is so that it is easy, programmatically, to ensure that the controlling program is in synchronization with the command parser in the propCAN device. Given this, then, we now add the additional operational requirement that when first starting communication with propCAN, we slowly send <CR>'s until we begin receiving them. Then we can start commanding the propCAN device with confidence.

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

THURSDAY, JANUARY 10, 2008, 23:58 MST

FIRST LOOK AT THE MCP2515-SPECIALIZED SPI ENGINE



When last we visited the SPI back-end of propCAN, we saw that the SPI demonstration code worked great but we can run and we need to run with an MCP 2515 specialized version of the code for the reasons we cited. This post describes the performance increase we are seeing with this first draft of the specialized engine.

The SPI code results described in this post are those attained with the object as posted to our Propeller Object Exchange at http://obex.parallax.com/objects/227/. Let's look at the new performance.

The captured trace (here it comes.... "Click to enlarge") shows a slightly different SPI transaction this time. The transaction is writing two bytes to the MCP 2515 and then reading a byte from it.

Let me back up a minute and present what we've done in the assembly code in this specialized engine then we'll review the numbers shown in the waveform.

This first draft of the engine implements the "entire command" model in assembly. When studying the overall MCP 2515 SPI command set, we find that we can accomplish all we need with roughly seven command types which correspond to:

- send 1 byte, read none
- send 1 byte, read 1 byte
- send 2 bytes, read 1 byte
- send 3 bytes, read none
- send 4 bytes, read none
- send 1 byte, send 13 bytes (yes, send then send)
- send 1 byte, read 13 bytes

Therefore, early in the assembly code you now see a jump table for these seven command forms. Also, in this first draft you see that control of the /CS line is now handled by the assembly code as well.

Let's get back to our diagram. The command we are using is the third form: send 2 bytes, read 1 byte. The /CS markers are (M5-violet and M6-cyan) which shows this command to now take only 11.62 uSec. We have improved, somewhat.

There is a slight difference in read versus write speeds so let's now look at these. The read-bit time is shown with the red markers (M1 dark-red and M2 light-red). These single bit reads look to be fairly regular (eye-balling the waveform) and the M1-M2 measurement shows this to be 440 nSec. (or a bit rate of 2.27 Mbps).

The write side is slightly faster and is shown with the green markers (M3-dark-green and M4-light-green). Here we see a bit time of 380 nSec (or a bit rate of 2.63 Mbps).

So, we have improved the handling of the SPI interface to the MCP 2515. I have carefully spoken to this being a "command handling" back-end though because of the final point we made in the earlier post. The hand-off time from one Cog to this dedicated SPI Cog is still slowing us more than we really want. So we will, even with the vast improvement shown in this very specialized SPI Engine, still need to move to "transaction" handling in the assembly code. That is small sequences of commands being handled within the assembly engine not just single commands.

Next post? Serial is working, the interrupt/status lines are working, and we've improved on the basic SPI performance. So in the next post I'll discuss an overall transaction (request to send a can message received via serial-Rx all the way to responding with a response CAN message sent back over serial-Tx) which will be _very_ telling as to what we next need to improve as we head towards meeting the desired performance of propCAN.

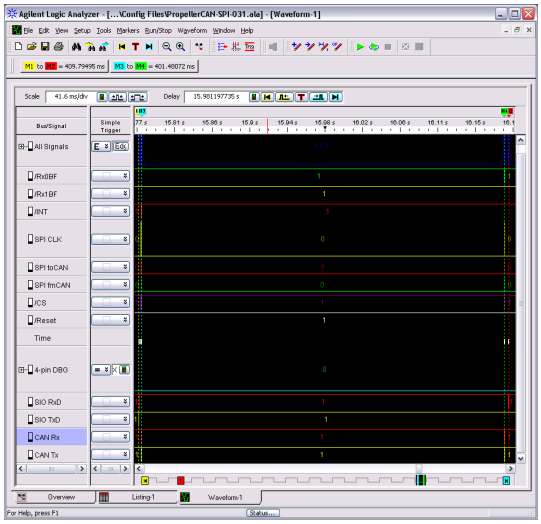
#MCP2515 SPI Engine

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

TUESDAY, JANUARY 15, 2008, 22:14 MST

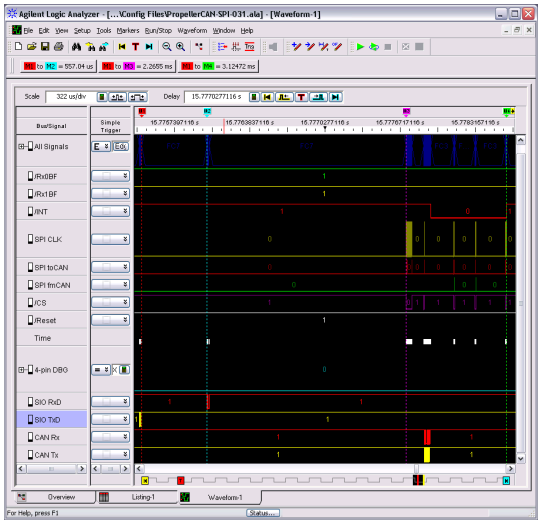
ANATOMY OF A FULL TRANSACTION - SIO-CAN-CAN-SIO



So here it is... (if you look closely ;-) At the left edge of the screen we have the serial data arriving at propCAN requesting a CAN message be sent. At the extreme right edge we have the CAN traffic arriving and then being formatted as ASCII and being returned via serial to the PC (over USB).

This is a fairly impractical view but let's look at a couple of numbers just before we zoom in on the left edge (then the right).

The view shows the request to response (M1 to M2) taking 409.79mSec with there actually being (M3 to M4) 401.48 mSec of "dead air" (nothing happening) in-between.



This second waveform shows the left edge where we can see the serial data arriving followed by the packet being sent to the MCP 2515 which is then followed by the packet being sent from the MCP 2515 via CAN. Let's look in more detail.

On the SIO RxD/TxD lines we see the request to send a message arriving (M1) and being acknowledged (M2). Next (at M3) we see the message being loaded into the transmit buffer of the MCP 2515 followed by a request to send the message now in the buffer. On the bottom two lines CAN Rx/Tx we see the message being transmitted over the CAN bus. We see the packet on both the Tx and Rx CAN lines because our own receiver listens to all traffic we send so that it can tell when it is allowed to send. (Collision Detection) and so that it can tell when other CAN devices acknowledge receipt of the message it sends.

If you look at the /INT line (MCP 2515 Interrupt request line) we see that immediately after the can message is sent the interrupt line is asserted. This is telling us that the transmission has completed and transmit status is now available (and the transmit buffer is now empty).

The next three SPI commands (1st two are (1) load tx buffer and (2) send tx buffer) are interrogating the MCP 2515 to determine health of the transmit, the last command is clearing the Tx complete interrupt.

Looking at the measured times we see that there is a lot of overhead while causing this message to be sent. A large portion of this overhead does not matter as a delay from when we choose to send to when the send occurs does not matter as long as we can send rapidly once we start to send.

However, a couple of general observations should guide our further efforts as we increase our device performance through code rewrites. These are

1. The time from interrupt assert to interrupt clear can be shortened and should be since this directly affects how rapidly we can send subsequent messages.
2. The time to generate a response on the serial interface and the time it takes to hand off the message to be written via SPI to the MCP 2515 are related in that one Cog is handling all of this. We can probably overlap these functions by splitting this effort amongst two Cogs.
3. A significant portion of the M1 to M2 response time is due to bytes arriving one at a time from the Serial Rx Cog. If these transfers were a <CR> delimited line at a time this would be much shorter.

Next post? Let's look at the right edge (CAN message arriving) followed by the end of the right edge where we finally see the ASCII interpretation of the message being sent to the PC via USB.

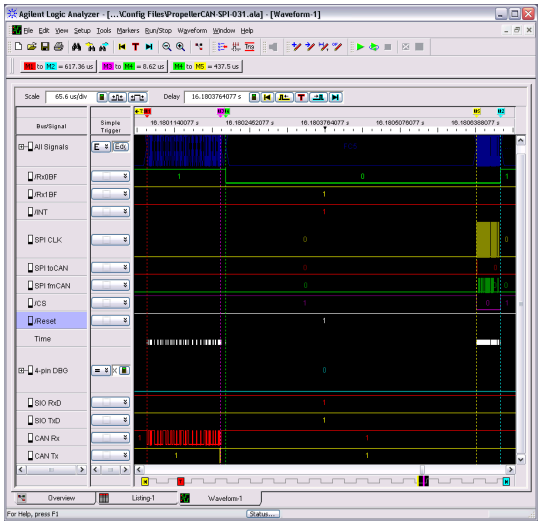
#CAN traffic, Serial, Serial Traffic

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

MONDAY, JANUARY 21, 2008, 18:59 MST

ANATOMY OF A FULL TRANSACTION - PART 2



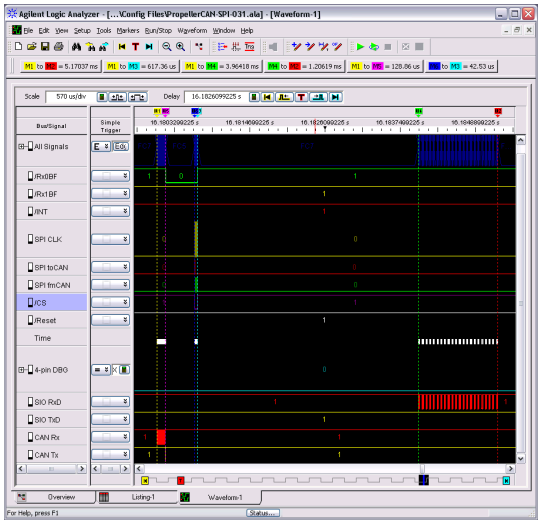
In the last post, we talked of the transmit side of things. In this post, we look at the receive side and the final sending of the received traffic to the PC via USB.

In the first waveform I've zoomed into the receiving of the message via CAN Bus and then the unloading of the message from the receive buffer within the MCP 2515.

Let's study the events in the first waveform. At M1 we see the message arriving over the CAN Rx line (in red). Then at M3, we see the MCP 2515 acknowledging the message by writing a bit in the ACK field (yellow CAN Tx line near bottom.) At M4, we see the interrupt line being asserted (green line, /INT, going low). After a period of time, we see traffic over the SPI bus. This is our firmware responding to the interrupt assertion and requesting the unload of the receive buffer. At the completion of the unload, we see the interrupt line be de-asserted.

I marked three times of note for this first waveform:

1. M1 to M2 (617 uSec) shows the time from starting to receive the CAN message to the time it has been unloaded from the MCP 2515.
2. M3 to M4 (8.6 uSec) shows the time from the message being completely received to the time the MCP 2515 asserts the /INT line.
3. M4 to M5 (437 uSec) shows the time it takes for the propCAN firmware to respond to the interrupt by unloading the received message.



In the 2nd waveform I've included all we see on the first but I zoomed out so that we can now see the received message being reformatted in ASCII and sent via serial to the FT232R chip which in turn sends it via USB to the attached PC. With this wider view of the receive activity we can now see the relative scale of our transmitting this same message over the various I/O channels. Our message is arriving by CAN Bus at 800kbps. It is being unloaded over SPI at a much higher rate and then finally we see it being sent over serial at a very slow rate, one character at a time.

A few of the times measured in this waveform are worth pointing out:

1. M1 to M4 (3.96 mSec) is how long it takes from then the message starts arriving at our receiver to when we start send it out in ASCII form over the Serial Tx line.
2. M4 to M2 (1.2 mSec) is how long it takes to send the ASCII message over Serial.
3. M1 to M5 (128.86 uSec) is how long it takes for the message to arrive over the CAN Bus.
4. M6 to M3 (42.53 uSec) is how it take to unload the message over SPI.

These waveforms show us how our current firmware is working and what transfer rates we are achieving over our various interfaces. In general looking at this post and the previous, we see that our firmware is taking way too long when transmitting over serial. We see, too, that we are taking too long when responding to the interrupts. Overall, all of our interfaces need attention.

Next post? Let's go over our internal message data storage format (messages in transit between interfaces.) and we'll go over what approaches we'll take as we make adjustments so that we can get as close as possible to our desired message throughput's.

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

MONDAY, MARCH 17, 2008, 19:18 MST

INTERFERENCE

Ah the best laid plans... I was actually thinking in late January that I'd be able to stay focused on this project. Alas, this has not proven to be so. But, I'm back... I'm spinning up now on where I left off and am deciding which aspects to work next. I'm actually getting anxious to get into propeller work again as I've been away too long. I've been reading articles and reading the Forum posts to wet my whistle for the project again. (deSilva, I can't thank you enough for all the posts and your articles that have brought me up to speed so quickly and for your always helpful posts to my questions on the Forum! You've been nothing but the best of help -IMHO. I sincerely hope you'll continue to lurk and offer us the benefit of your expertise. Good luck in your ongoing endeavors.)

Watch for new posts to start happening here next week if not late this week. I wonder what I'll work on first??? ;-)

#Progress lack thereof

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

WEDNESDAY, MARCH 26, 2008, 00:47 MST

INTERNAL MESSAGE STORAGE FORMATS

Well hi there! It's good to be having fun working on this project once again!

When last we discussed the internals, we were musing at all the "tuning" ahead of us yet in terms of the performance of our existing "proof of concept drivers". It is this driver tuning which will be the subject of my next posts but let's get into how we are choosing to move data through this system in this post. Our reducing data movement and transformation is our first step in tuning this system's performance.

Our goal is "best possible throughput" and to meet this we must be efficient. This means we want a minimum of format translations and a minimum of copying data from one location to another within propCAN. Let's look at our USB and CAN interfaces to see in what natural forms our traffic exists and this should lead us to defining what storage forms we will use. First let's look at the USB side.

Messages to be sent via CAN arrive as ASCII strings via USB. These strings are anywhere from 5 to 22 bytes for standard mode (11-bit ID) CAN packets and 11 to 27 bytes for extended mode (29-bit ID) packets. In addition to messages, commands to propCAN and status strings from propCAN are also sent via USB but these are all smaller than our 27-byte longest message. So this then is our USB side definition:

- USB I/F: strings are sent and received which are 2-bytes to 27-bytes in length.

Now let's look at our CAN side. The MicroChip MCP 2515 receives packets into 13-byte buffers with an additional (optional) byte of status at the end. Both our standard and extended messages fit into this size buffer. In order to send a message the 2515 also wants the message formatted into this layout for best efficiency hand-off from the Propeller to the 2515. Also, we have an additional requirement to time-stamp arriving messages (relative time from packet to packet) so we'll add another two bytes to this length. So here we have our CAN side definition:

- CAN I/F: messages are in 14-byte buffer with two byte time-stamp (16-bytes total).

If we round our needed buffer sizes up to multiples of longs (our largest native Propeller data size) we end up with 32-byte buffers on the USB side and 16-byte buffers on the CAN side. Let's look at the resulting minimum transforms we end up with:

- SEND side: a message arrives as an ASCII string received by our Serial Receive COG. It transfers the message into a 32-byte buffer in Main RAM. A pointer to this 32-byte message is handed eventually to validation and then conversion code. This conversion code grabs a 16-byte buffer, again in Main RAM, and re-formats the ASCII text into an MCP 2515 layout message.

The pointer to this 16-byte buffer is then passed to the SPI send routines at which time the buffer is accessed by the SPI cog and bytes from it are sent to the MCP 2515 device. Since the layout of the 13 early bytes in the buffer are exactly what the 2515 most wants the transmission to the 2515 is efficient: command followed by 13-bytes of data (actually, less bytes need be sent if the message payload is less than the full 8 bytes.)

- RECEIVE side: this is pretty much the same effort but in reverse sequence. The 2515 receives the message and then sends an interrupt to the Propeller. Upon receiving the interrupt a 16-byte buffer is populated with the data offloaded from the 2515. We append the time-stamp of the unload effort as the last couple of bytes and the rest is the reverse of the send side effort.

Now we are beginning to understand the data flow through the propCAN system. We are not moving data around or reformatting it unless we need to. We've normalized our system buffers in Main RAM to 16 and 32-byte objects a convenient size for the Propeller.

Next we'll look at an approach to getting our serial drivers to be much better performing.

#Message formats USB CAN traffic storage

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

SUNDAY, MARCH 30, 2008, 00:25 MST

PERFORMANCE A QUICK REVIEW

I've not posted since Wednesday because I've been off writing new propeller objects. Both serial transmit and serial receive were limited to 460,800 baud after my last rewrite of them. PropCAN needs to be able to run at ~1,500,000 baud if we want to meet best possible rate of messages arriving from the CAN bus. These numbers motivated the re-writes. Let's quickly review two of our key performance criteria.

The AMSAT CAN bus runs at 800,000bps. We have three CAN message sizes in the protocol: 0-byte payloads, 2-byte payloads, and 8-byte payloads. If the bus traffic was the maximum load of each of these messages the traffic rates would be as follows:

- 0-byte payload: min. of 47-bits per message, max 17,021 messages / sec.
- 2-byte payload: min. of 63-bits per message, max 12,698 messages / sec.
- 8-byte payload: min of 111-bits per message, max of 7,207 messages / sec.

PropCAN translates these messages into ASCII strings and sends them up the USB interface to the connected PC. After formatting in ASCII let's look as what happens to the message size:

- 0-byte payload: **tiiiL<CR>** (6-chars x 17,021 /sec = 1,021,277 bits per sec.)
- 2-byte payload: **tiiiL0011<CR>** (10-chars x 12,698 /sec = 1,269,841 bits per sec.)
- 8-byte payload: **tiiiL0011223344556677<CR>** (22-chars x 7,207 /sec = 1,585,586 bits per sec.)

OK, that's a lot of numbers but it shows us something interesting. It shows us that two message formats cause different ends of PropCAN to be working the hardest (at the fastest rate) and NOT at the same time. The 0-byte payload messages (the shortest ones) make our SPI offload run at maximum performance (1.8 Mbits/Sec) but the demand on the Serial Transmit routines is lower. However, the 8-byte payload messages don't load the SPI offload routines as much but now our Serial Transmit routines must run at 1.5Mbps! And now you see where I came up with my ~1,500,000 baud quoted in my opening paragraph.

Now we've reminded ourselves of Max SPI offload rate we need, the max message handling rate we need and the maximum Serial transmit rate we need. Our goal is to not let our I/O performance run slower at some point in our design which would cause us to need to buffer messages and then, ultimately, to not keep up with traffic arrival rates. We know we've got work to do to make this system work at our needed performance.

Now that we've clarified our goals, let's move on to the topic for the next post: my now working code for the new Serial Transmit and Receive Objects.

#Performance, Serial Traffic, USB I/O

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

SUNDAY, MARCH 30, 2008, 01:05 MST

CIRCULAR RING OF BUFFERS FOR SERIAL IN/OUT - FAST!

In reviewing the waveforms (Logic Analyzer capture pictures in previous posts) we note that we are passing a character at a time from our serial receive Cog to its listening Cog which is the wost performing of the Transmit and Receive sides. This hand-off originally limited the code to running at 230,400 baud. After some instruction choice changes and making it more specific to interacting with the FTDI FT322R chip I was able to improve this to working fairly reliably at 460,800 baud but no better. My goal in re-writing the Serial objects is to reduce this cog interaction to needed boundaries thereby allowing the serial transfer rates to be much faster and to, hopefully, to attain our needed 1.5M baud.

Our study of the serial traffic has shown us that this application has string length maximums and an overall performance need. Let's make use of this knowledge and only transfer from Cog to Hub on each long within command (max string length) boundaries, not character boundaries. In the highest traffic cases we will then have 6 transfers were were seeing 22 transfers earlier. Let's also not make our Cogs wait on each other by providing more than one buffer so that a buffer can be being transmitted while another is being filled. Likewise, a buffer can be filled by the receiving Cog while another is being acted up by the command processing Cog.

In our serial transmit case we can transmit a buffer at full speed and not break stride until we next need another buffer. If another has been prepared we simply switch to it and start sending it; again at full speed.

In our serial receive case the original object was limited by our ability to dump characters into Main RAM. This severely impacted our ability to be ready to receive the next character. Since we now know that our characters streaming at us are broken into max command/message size strings,we can defer writing into Main RAM until we hit one of those natural boundaries where we have more time. This means that we can receive serial data from programs controlling the PropCAN at full serial rate without missing characters!

The command protocol for PropCAN ensures that activities are gated by acknowledgements which means that only one or sometimes a couple of commands will arrive before an acknowledgement must be sent at which time the sender will stop sending and wait for a response. This is great in that it allows our new routines to perform well but not reach the limit in performance they too have.

These new routines are built around a new data structure the contents of which controls the actions of the single producer and single consumer of data contained within the structure. The new data structure

is simple. We have an array of pointers to fixed size buffers. The number of pointers in the array is adjustable. The fixed size buffers contain a length/flag byte and the rest of the space is for the data (to be received or to be transmitted). The fixed length of these buffers is the same for all buffers pointed to by the array but is adjustable. For the PropCAN device we use 32-byte fixed length buffers and the array pointing to them consists of 4 entries pointing to four unique 32-byte buffers. We choose the 32 because it is the next power of two greater than our maximum 22-character command/message. We chose the 4 rather arbitrarily (read- based on "no real data") but it can be adjusted separately for Transmit and for Receive as we discover what our real depth needs to be. The Transmit side and the Receive sides each have their own independent data structure instance (array of pointers and set of buffers to which they point.)

Did I catch you on my having an array of pointers to fixed length buffers? Isn't a concatenated set of fixed length buffers also an array? Why have the array of pointers to the array of buffers? The answer is really quite simple. It's the old standard trade-off between memory and performance. Think of this array of pointers as a pre-calculated set of answers. With the array of pointers I have much less code and therefore much less execution time in calculating which buffer will be used next. I simply move the preparation of these pre-calculated answers out of the critical path of when i needed to access buffers to when I'm starting up the PropCAN device; a much less time critical point in time.

I mentioned that our length byte in each fixed buffer is really length and flag (dual purpose). Let's look at why I think this. The array of pointers to these buffers lets us easily treat the set of buffers as a circular list. So we'll let the producer start at array[0]'s buffer and when it is filled the last value written will be the length byte. The consumer will also start at array[0] and will not consume the buffer until the length byte becomes non-zero. When the consumer is finally done with the buffer it zeros out the length byte and moves on to array[1]'s buffer and waits for it to have a non-zero length. Likewise, our producer sets a length in array[0]'s buffer and then moves on to filling array[1]'s buffer. As each tried to locate the next buffer after finishing the last in the array (in this case array[3]) then they wrapped the index back to zero and started again with array[0]'s buffer. So we have the buffers being used in a circular fashion and we have the length field within each buffer being used as a "buffer is empty - can be filled" or "buffer is full - can be emptied" flag.
Well, it's time to end this post but first let me describe the state of the code since my Wednesday post. I've implemented and tested the new "circular-fixed size buffer" handling Serial Receive and Serial Transmit objects. I've run them at baud-rates from 19,200 to 923,076 baud rates without error or data loss. Tomorrow I'll update my FTDI driver installation on my Windows XP PC so I can test them at 1.5M baud and 2M baud (I know they won't run at 3M baud so I won't go to the FTDI 3M baud during my testing...<sigh>)
My next post will be showing Logic Analyzer waveforms of the areas shown in earlier posts were we saw the serial traffic taking so long. We hope to see the serial traffic no longer dominating the waveform as it was in the past. We hope now to be taking up only as much time as we really need!
#Data Structures, Performance, Serial Traffic, USB I/O

turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

MONDAY, MARCH 31, 2008, 00:58 MST

NEW SERIAL DRIVER TESTING RESULTS

As promised, I've a new waveform to speak to today (yes, actual measurements!) but I'll get to that later in this post.

I spent my time today testing and making some late adjustments to code so that it would run correctly. As you remember in my last post I noted that I was running Rx and Tx at what Windows calls 921,600 baud and it was working well. However, next I had to modify my Windows FTDI driver installation to get to higher baud rates so I could continue testing.

After a few false starts (read- two driver reinstallation's and a first incorrect attempt at modifying the 3 alias baud rate entries...), I managed to get my FTDI drivers configured correctly. What's fun is how I know I did. In my last efforts yesterday I upgraded my interaction with the auto-baud routines to show detailed findings when asked and I also added a sign-on transmission which happens immediately after the auto-baud completes. This is where it gets fun! I'll show you.

First, here are my speed aliases for my updated serial drivers:

- 300 baud -> 3M baud
- 1,200 baud -> 1.5M baud
- 2,400 baud -> 2M baud

I chose these aliases so that I would have some hope (in my older age ;-) of remembering the new alias values. I also have no desire to be running this device at those original baud rates.

When I first start PropCAN now and send it a $0d <CR> is responds with the following message:

* PropCAN connected at 923076 baud, (92307 chars/sec) *

(It's pretty easy to tell when things are working well enough for it to send this <grin>.)

So I select 1200 baud (intending to get 1.5M baud) and connect to PropCAN and I see the following:

```
* PropCAN connected at 1500000 baud, (150000 chars/sec) *
```

I'm smiles all over!

So I ask it it's current configuration by entering ?<CR> and get the following:

```
D PropCAN - A Propeller-based USB to CAN Controller by KZ0Q
D0 F/W Debug: OFF
VA201
O? CAN Closed
S7 CAN at 800 Kbps
X0 AUTO Fwd: OFF
Z0 Time-stamp: OFF
```

I move on to 2400 (new 2.0 Mbaud) and try once more:

```
* PropCAN connected at 2000000 baud, (200000 chars/sec) *
```

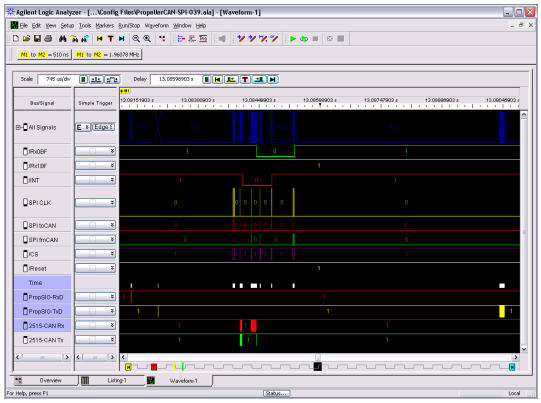
Now I'm grinning ear to ear.. but then I entered the ? command and got nothing. Then I'm moved up to 300 baud (3M baud) and I got nothing... Hmm, this is an amazing start to my testing but I've got work to do.

Now remember, I've an analyzer connected which is currently sampling at a 5 nS rate so I can pretty much accurately determine if what is being sent and received is actually at the bit rates I think it should be at. I verified each of these speeds and yes the driver is correctly sending at the 1.5, 2.0 and 3.0 bit rates!

This is where I had to go back and adjust things. My transmit side needed some code tweaking as it was exhibiting some really fat start-bits at these three speeds. I was able to make better instruction choices to bring this into alignment really well. On the receive side it took some studying. However, eventually I replaced a couple of test and jmp loops with waitpne/waitpeq and the receive side snapped to as well. After these changes, I was able to fully interact at 2.0M baud and I also retried my 300 baud ;-) test with the following results:

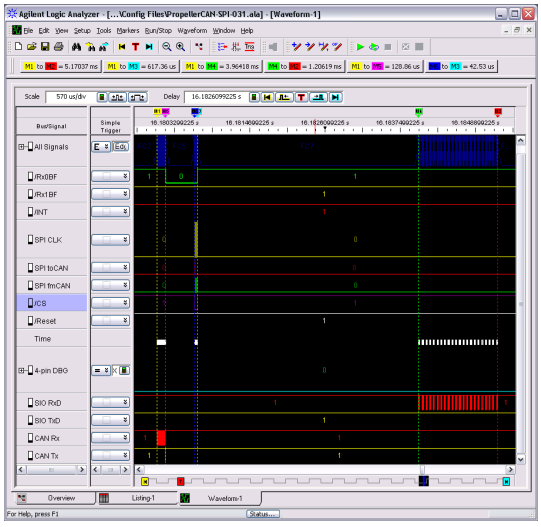
```
* PropCAN connected at 3000000 baud, (300000 chars/sec) *
```

(No I cannot interact with it at this speed, tho' I did try... I also knew by my calculating the instruction timings of my code that it would not work. This is why I posted the question I did at our Propeller Forum...)



So, in the end, this has been a great day. Now let's look at what's happened to our overall timing picture. I measured a full can transmit and the reply of the CAN Node's response sent back via USB. (I've no markers in this picture, so I'll talk in signal color and location instead...)

At the left, just up from the bottom we see to signals (red and Yellow) which are labelled as PropSIO RxD and TxD. I make the red color receive so these are easy to remember. The red descending blip at the left is the packet in ASCII coming in over serial "**T1430<CR>**" and the yellow blip following it is the <CR> reply back via USB. Moving on to the bottom two lines (red and green this time, 2515-CAN labels) we see our CAN message transmission in green and we see our CAN receiver seeing our full outgoing message (our receiver always sees our own transmissions in the MCP 2515.) Then we see the CAN Node replying to the message with a longer (8-byte payload) message. This is the next red burst. Just near the end of the red burst you see a green blip. This is the MCP 2515 ACK'ing the message (as each listener does on the CAN bus.) Now, look all the way to the right and you see the ASCII representation of the 8-byte payload message being sent to the PC via USB: "**t3628626362372604003 4<CR>**". Finally, and less important but so you know, the burst of activity above the CAN signals we just spoke of is the SPI traffic with interrupts.



The important part of this is seeing the relative time now taken for each of these actions. Just "eyeballing" the picture you can see that our serial transmission (in Yellow, and at 2M baud) is about the same width (takes about the same amount of time) as the CAN message arriving from the CAN Node (in Red). Contrast this against this 2nd picture originally shown in the Jan 22 post "Anatomy of a full transaction - Part 2". In this older picture we see the serial transmission burst (this time in Red) again to the right but look how much time it was taking! That's 1.2 mSec! vs. the 128.9 uSec of the CAN message arriving.

I think you'll agree this is exactly the improvement we've needed to meet our timing goals for this project. We needed 1.5M baud and we've got a functioning 2.0 M baud giving us timing margin and in fact our transmit side works to 3.0M baud according to our testing.

This has been some fun work on serial drivers for PropCAN! We see the auto-baud object working well at all speeds of the FT232R. We see the serial transmit object working with all speeds as well. And finally we have the serial receive object working at the supported speeds up through 2.0M baud. This is all but the highest 3.0M baud speed. We've done ok.

Now it's time to work on the next performance upgrade for the SPI subsystem. I'm off to study what to do next there... I'll be posting more later this week after I've made some progress.

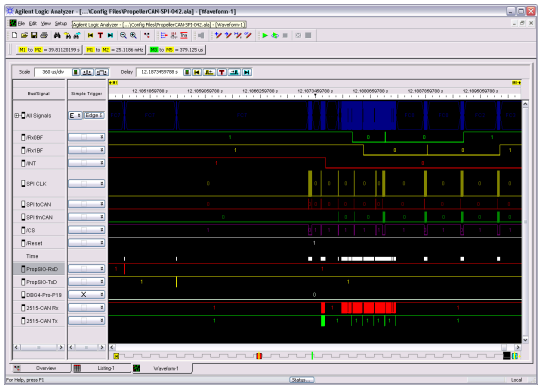
turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

MONDAY, April 7, 2008, 23:30 MST

IMPROVING SPI PERFORMANCE - OUR STARTING LINE

OK, I've spent the past week first discovering just what the current SPI performance is relative to live CAN traffic followed by determining how to address the issues and finally by implementing the solution. The exciting news is that it was a very productive week. It appears that the performance is now where it needs to be but further testing must be done to prove this.



Let me show you what I found the performance to be. The picture to the right shows us a number of things (most of which are not good ;-)

First let's look at how we initiated the traffic. I'm setup with 5 CAN nodes on the CAN bus. I sent a message out to the five CAN Nodes to have them identify themselves (part of the AMSAT protocol). Locating this in the logic analyzer trace, we first see the serial RxD (Red) traffic at the bottom left which is followed by the serial TxD (Yellow) "OK" response. Next we have the MCP2515 transmit buffer load (SPI-labelled signals in middle) and the send buffer command again via SPI. Then we see the actual CAN transmit (Green) signal at bottom followed by the CAN receive (Red) with the Green CAN-Tx ACKs for each CAN message arriving.

Notice how quickly these messages arrive? CAN is a great little protocol in that all devices listen to all traffic on the bus and they inject their message as soon as they can. They will wait only long enough (mandatory gap between successive messages) and then the next message is transmitted. This means that as a receiving device we wait around for traffic which will come in bursts and sometimes, like this test case, our traffic will arrive in maximum speed bursts. How fun! Well, OK it's not fun at all. This means that our device has to immediately support running at best possible speed. The only freedom we have is that the MCP2515 does/can double buffer received traffic but this is very little additional freedom.

Now, turning our attention to the top signals we see the /INT line (Red) being asserted which is our transmit complete signal to the Propeller. We then see the /Rx0BF signal (Green) asserting followed by almost exactly one CAN message later the /Rx1BF signal (Yellow) asserting. These are the "receive buffer zero full" and "receive buffer one full" signals. This is all working exactly as we want. We are being notified of each event as we intended!

Where we see the first indication of performance issues with our prototype code is that we don't begin to unload the 2nd message until after all 5 messages have arrived. (the first unload occurs during the last of the 4th message arriving and the 2nd unload occurs quite a bit after the last message arrived.) Given that we know we only have two receive buffers we just proved that we lost 3 of the 5 messages arriving. We just are not fast enough.

Our transmit complete clearing of the interrupt can happen much faster (should likely happen before the first message arrives) and certainly our receives need to be much faster. In fact this almost certainly proves that we can't have a separate Cog watching the /Rx0BF and /Rx1BF lines and then asking the SPI back-end Cog to unload the buffers as we do here. This simply isn't fast enough.

This means that our initial functional decomposition and assigning of responsibility amongst Cogs is not going to work. It looks like we have to move some of the transmit acknowledgement handling and most if not all of the receive handling. Well, it's back to drawing board for me to figure out which Cog needs to do what, one more time...

In my next post I'll describe how things are to get rearranged and then I'll follow up with measurements of the new organization.

#MCP2515 SPI Engine, Performance, SPI

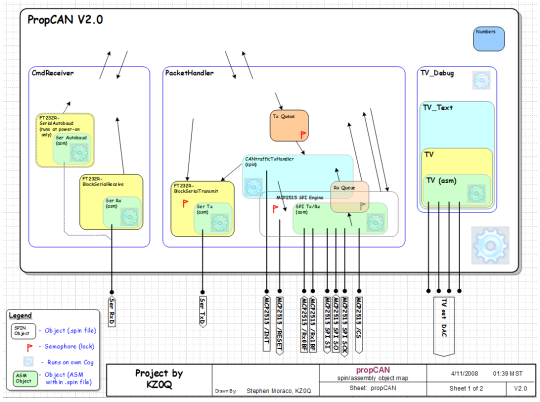
turning on propCAN

Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

FRIDAY, APRIL 11, 2008, 01:47 MST

OBJECT REORGANIZATION TO ACHIEVE SPI PERFORMANCE

I finally revised the PropCAN object diagram so I can continue this SPI performance discussion! ;-) Last post I identified why the CAN receive system had to be reorganized- We found that the draft organization was too slow to unload the CAN messages as they arrived!



This new object diagram may be confusing but I will try to sort it out in this post. (for the **original diagram refer to the January post entitled "Which Cog is doing what?"**) New in this diagram is the appearance of the TV Debug object-set on the right. The A.2 hardware has the four debug pins carefully chosen so that they can be used to drive the TV-out signals. I even created a little daughter card which contains the DAC resistors and the RCA connector attached to a socket header so it simply plugs into the 4-pin debug header.

(If you are interested, pictures of the hardware can be found at the project web-site: http://propcan.moraco.us in the "PropCAN Detail", "DEMO Board", and "Gallery" sections.)

Let's start this review by going over the list of significant changes between the v1.0 and the v2.0 diagrams:

- /Rx0BF, /Rx1BF pins now route to the ASM portion of the SPI Engine
- /INT now routes to both the SPI Engine and to the CANtrafficTxHandler
- The RxQueue now straddles ASM portion of SPI Engine and the CANtrafficTxHandler
- CmdReceiver fully encloses Serial Receive functions while PacketHandler fully encloses the Serial Transmit and the SPI Transmit/Receive functions

Now let's review the reasons why I made these changes.

The Rx buffer full pins now route only to the ASM portion of the SPI Engine so I could move the entire receive function into the ASM code. This is also the reason the RxQueue is overlapping the ASM portion of the engine. Not only did I move all receive but I also rewrote the receive so that the MCP 2515 buffers are unloaded directly into the Queue data structure. There is no longer any middle-man routine transferring unloaded buffer content out of this SPI engine and into the Rx Queue. We were suffering CAN messages arriving faster than we could unload them. We have to be **much faster**.

The /INT line is now handled in two areas of the code but for two different reasons. The SPI engine watches /INT to know when it can get Tx-complete statuses and when it needs to clear the Tx-complete interrupt. CANtrafficTxHandler receives /INT so that it can deal with Rx errors.

Lastly, I moved to more fully enclosing objects with minimal interface facades for the enclosed objects so that code outside of the enclosure is simpler and easier to read/understand.

Well, now you see what reorganization we did. Next post, I'll show measurements of the surprising performance increase these changes have yielded.

#Software Organization

turning on propCAN

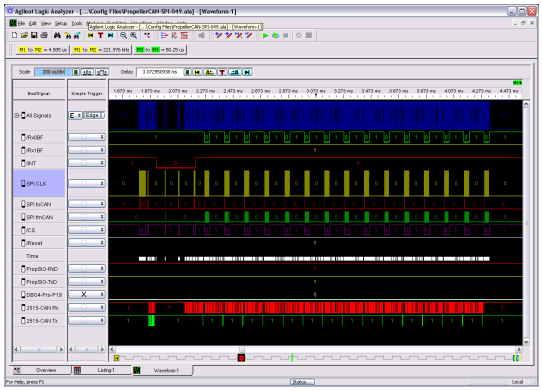
Developing a USB to CAN controller from the ground up. New multi-processor cpu, new language for embedded code, 1st time using this toolset. Thank goodness that I have good instruments and instrumentation for verifying behavior so let's get started...

SUNDAY, APRIL 20, 2008, 22:13 MST

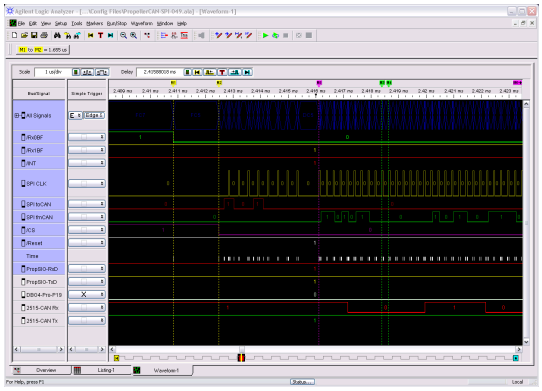
HOW FAR WE'VE COME... FULL SPEED CAN TRAFFIC!

In the last post I presented the new object organization. In the end we had to move responsibilities around (amongst cogs) in order to get the system to be as responsive as we needed. I started with simple symmetry and had to abandon it. At some level this surprised me, but our goals had to be met, so here we are.

In this post I've a bunch of waveforms to present. The good news is that each zooms in on a rather simple observation so there is not much detail to each of them but the resulting behavior is too fun to not show.



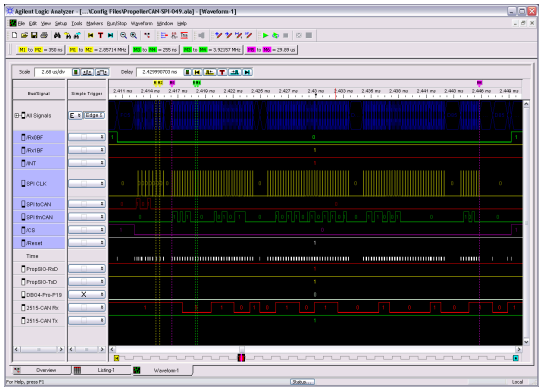
To the right is an overview of a Bus Census with something you haven't seen here before. This is the Census being transmitted from PropCAN (as directed by the host software, CDNC) and fifteen, yes 15!, CAN Nodes all eagerly responding. Do you see what happened? Remembering the last post we saw that we couldn't handle 5 nodes responding, but look at this. Here we have 15 nodes all responding as fast as the CAN Bus will allow and the new SPI Engine receive side is so fast that we never even needed our 2nd receive buffer. The /Rx0BF line (bright green, near top) show the signal being asserted and then our SPI Engine back-end Cog is so fast that we empty the buffer while we are still receiving the next message. It is wonderful to see an idea pan out, isn't it? Let's quickly walk thru a few significant parts of this traffic which we'll do by presenting four more pictures.



In this next picture I've zoomed in on the asserting of the /Rx0BF line (buffer full for the 1st message to arrive.) I've placed markers M1 and M2 at the time buffer full is signaled and at the time our Rx-side begins to send the buffer unload command. I've placed a time interval measurement label at the top left. We can clearly see that M1 to M2 = 1.655 uSeconds (or 33 assembly instructions.) Our SPI Rx back-end Cog was in a loop deciding what needed to be done next when it saw the /Rx0BF line being asserted at which time it formulated the

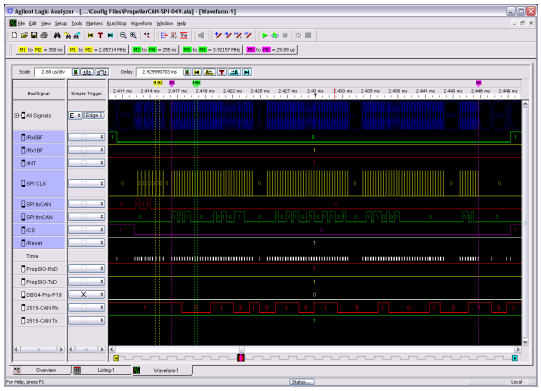
command to unload the interrupting buffer and initiated sending the command via SPI. Not a bad response time.

As long as we have this view, let's look around. The other waveforms quickly show us the relative speeds of each of the communication happening at the same time as our unload. From top to bottom, we see the SPI CLK signal showing us our SPI Write speed as we see the command to unload the buffer being sent to the CAN controller (MCP 2515). Below the SPI CLK we see the SPItoCAN which is our command data being sent and below that is the SPIfmCAN signal which is data returning from the 2515 (our received message arriving). Finally, at the bottom we see 2515-CAN Rx toggling. This is our next CAN Message arriving while this buffer unload is occurring. Compared to our SPI Tx and Rx rates the CAN bits seem pretty slow. We remember that our CAN rate is 800,000 bits/second. What is our new SPI Tx/Rx rate? Let's look at our next picture to see.



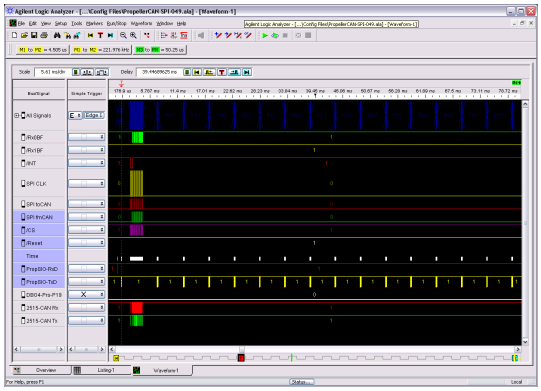
In this next picture we've shifted to the right along our unload so that we can clearly see the SPI Tx and SPI Rx bit rates. The code is asymmetrical for the similar reasons that our Serial Rx and Tx turned out to be asymmetrical. The work we are doing is different for the two sides of the effort. Anyway, I've placed two markers for Tx and two more for Rx and I've placed two measurements for each which tell us the duration and bit-rate that we are now seeing. Let's look at them.

Markers M1 and M2 are spanning a single byte write from the Propeller to the MCP 2515 device. At the top we see that this write is 350 nSec long, resulting in a 2.857M bits/second (bps) SPI transmit rate. On the receive side, Markers M3 and M4 bracket a single byte being received. The duration of this byte is 255 nSeconds and the resulting bit-rate is 3.922M bps. We are commanding the CAN device at nearly 3M bps and receiving data at nearly 4M bps. If you remember our performance criteria we set for ourselves these speeds are better than we needed to meet our goals. At this rate, and since we are getting it done during the receive of our next CAN message just how fast are we unloading the receive buffer from the MCP 2515?



In this fourth picture I've zoomed back out a bit so that we can see the entire unload. I've placed new markers (M5 and M6) along with a new interval measurement. M5 is at the start if the first received byte while M6 is placed at the end of the last byte emptied from the receive buffer. The interval measurement shows that we are unloading the buffer in 29.89 uSeconds. Finally, just shortly after the last byte is read, the /CS is de-asserted and the /Rx0BF is also de-asserted which

happens automatically when we unload (read) the receive buffer. Now, lets zoom out one last time to look at the full transaction again but let's zoom far enough to see the serial transmitting of the messages to the connected PC via USB.



Here in this final picture you can see the overhead of handing off the messages between cogs and finally getting sent over USB as the dominating feature of the waveform. You can see what I mean when you see that all of our 16 CAN messages (1 outbound, 15 in-bound responses) are gathered in the colorful burst at the left of our waveforms. Then, spread out, over the remainder of the picture to the right of the burst we see the yellow message transmits over serial from the Propeller to the FTDI chip to be sent over USB to our PC. While each message is very fast (the yellow burst is narrow) there is a lot of dead time between each message. Depending upon the rate of messages arriving via CAN this could spell doom for our system if we don't have enough buffering to handle the majority case. That is enough space in the Rx Queue so that we can place arriving message from CAN into the Rx Queue and then hold them there until they are sent via serial to the FTDI chip and the entry in the Rx Queue then freed up for use by another arriving CAN message.

So, in the end, we've got system that's performing much better and so far is meeting our basic performance specifications. Now it's time to move on to quantifying the performance of PropCAN under normal system loads and with normal traffic patterns. We're almost there!

#CAN Traffic, MSC2515 SPI Engine